EECE5644: Assignment #1

Machine Learning and Pattern Recognition

Name : Nikita Vinod Mandal

NUID : 002826995

Question 1 :

Part A – Minimum Expected Risk Classification

Using the mean and covariance matrices given, 10000 samples are generated having multivariate gaussian distribution.

1. Minimum Expected Risk Classification Rule is as follows:

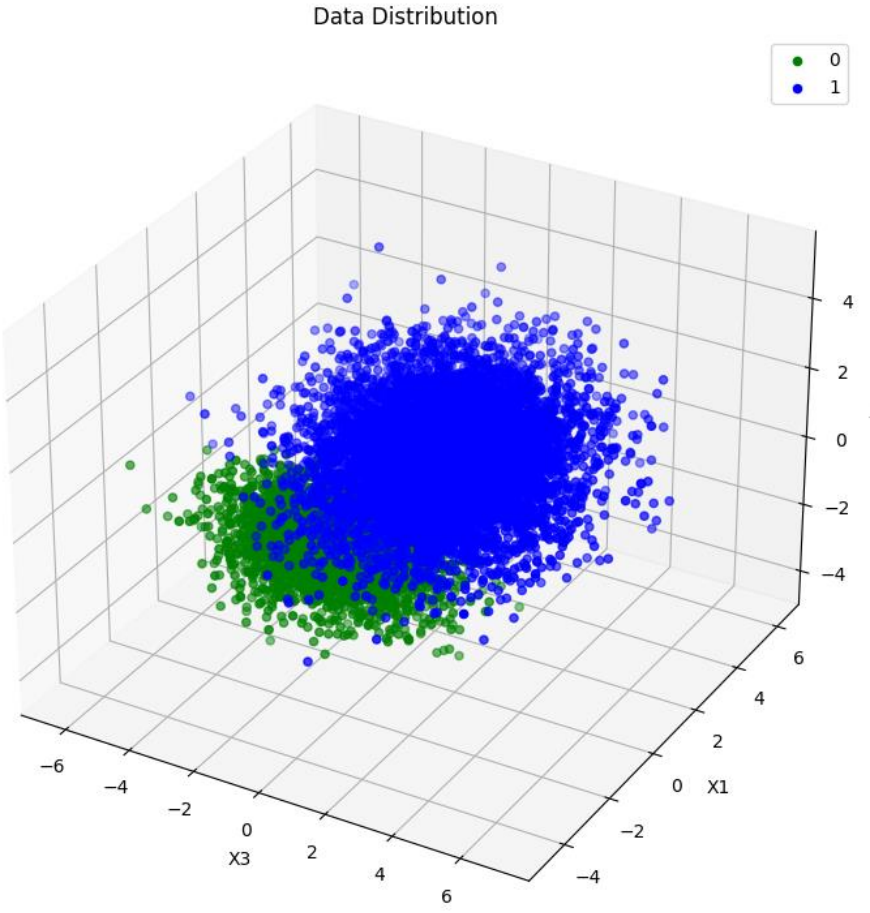
$g(x|m0, C0) > P(L = 0) (\lambda 10 – \lambda 00) = (0.7) (\lambda 10 – \lambda 00)$

$g(x|m1, C1) < P(L = 1) (\lambda 01 – \lambda 11) (0.3) (\lambda 01 – \lambda 11)$

$(D = 1)$

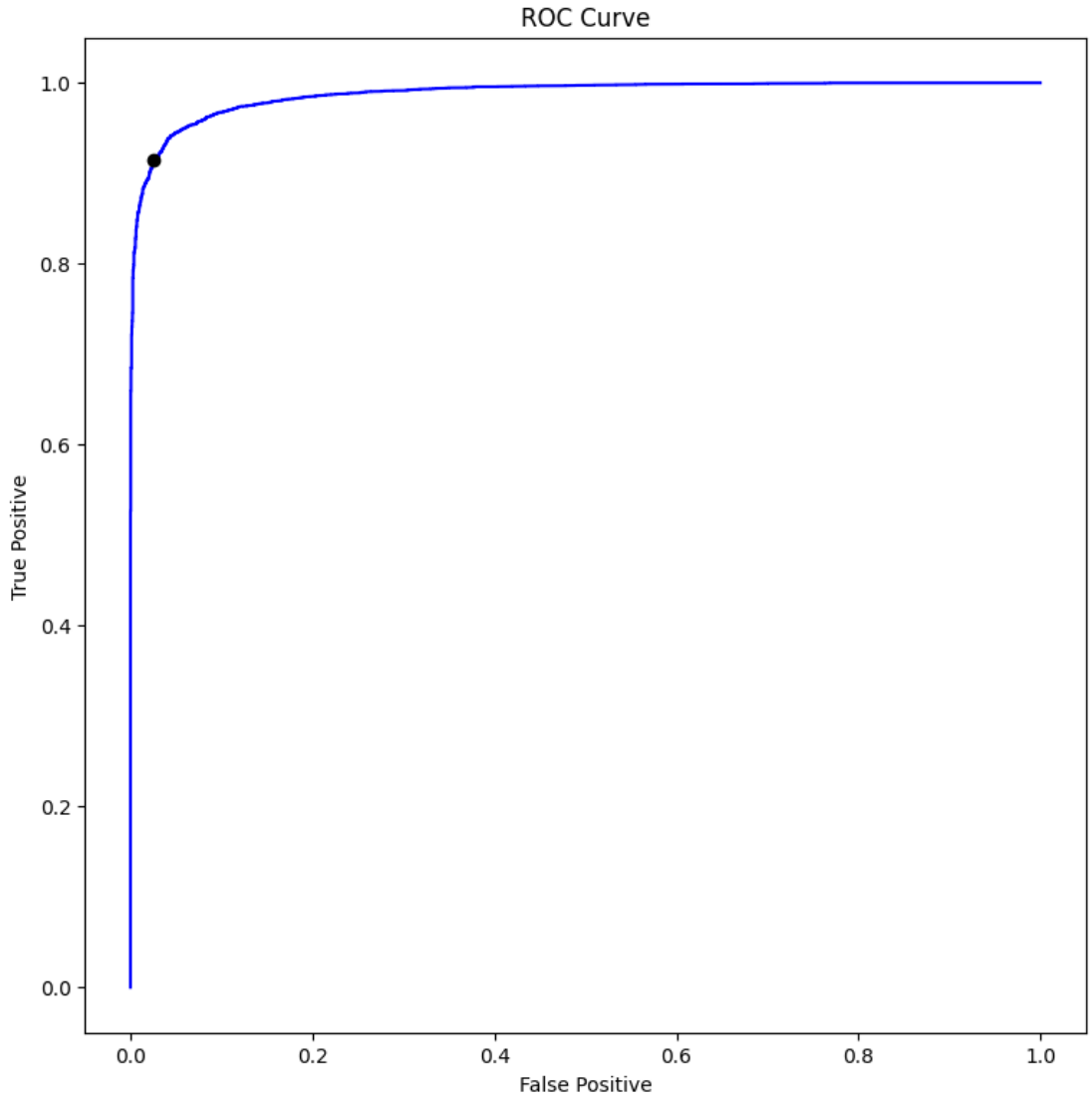
$g(x|m0, C0) > (7) (\lambda 10 – \lambda 00) = \gamma$

$g(x|m1, C1) < (3) (\lambda 01 – \lambda 11)$



$(D = 0)$

2. The ROC curve is generated by approximating the variation of threshold value γ from 0 to ∞. Practically, γ values are sorted and the mid-points between consecutive two values are considered as threshold points ranging from minimum to maximum.



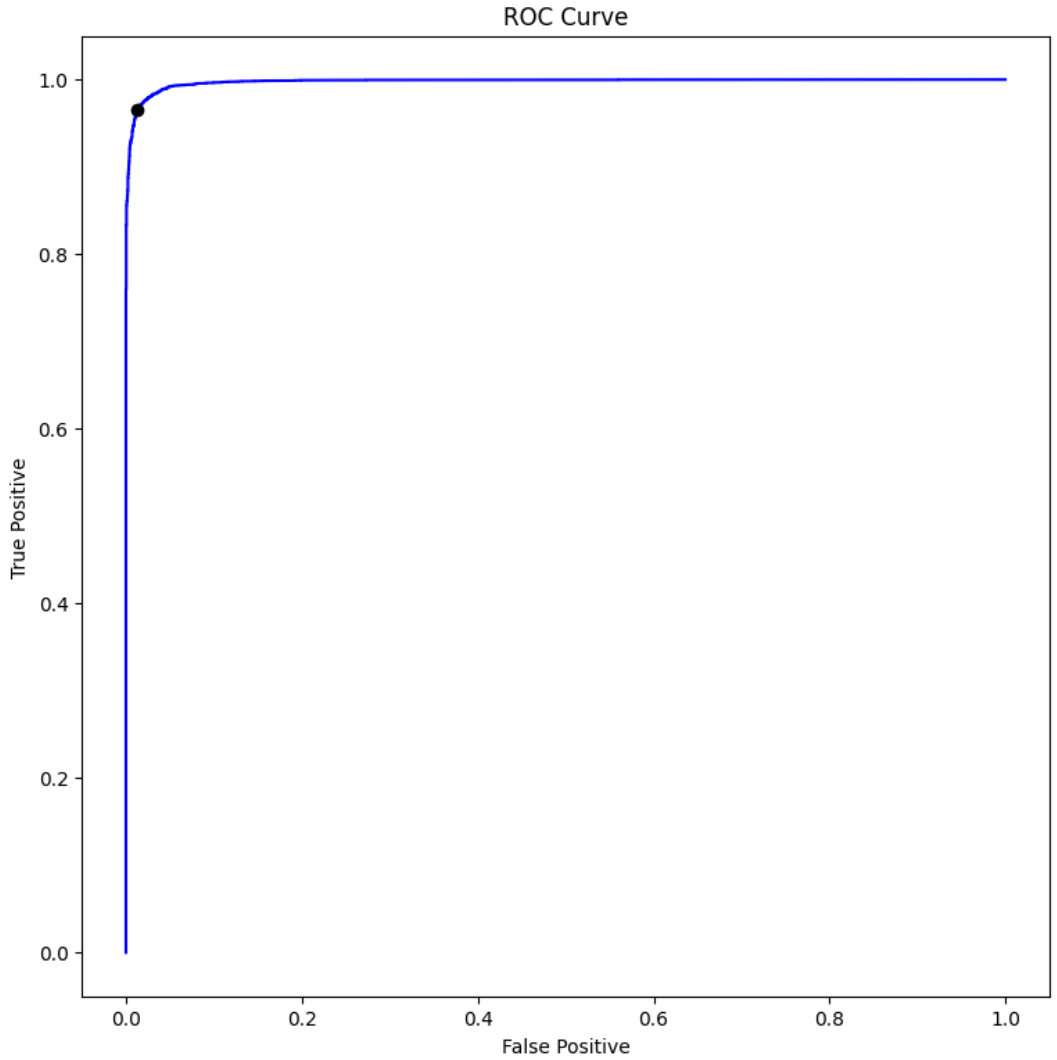
3. Theoretically, the value of γ is found out by dividing the class priors (0.7/0.3) which is equal to 2.33. With this threshold, false positives and true positives are computed which help in estimating the Minimum P(error). It can be observed that there is negligible difference in the theoretical and experimental values.

Gamma Ideal - 2.333333 and corresponding minimum error 0.044522
Gamma Practical - 1.568752 and corresponding minimum error 0.043946

Part B – Naive Bayes Classification

1. Minimum Expected Risk Classification Rule is as follows:
gNB (x|m0, I) > P(L = 0) (λ10 – λ00) = (0.7) (λ10 – λ00)
gNB (x|m1, I) < P(L = 1) (λ01 – λ11) (0.3) (λ01 – λ11)
(D = 1)
gNB (x|m0, I) > (7) (λ10 – λ00) = γ
gNB (x|m1, I) < (3) (λ01 – λ11)
(D = 0)

2. The ROC curve for Naive Bayes Classifier is not as sharp as that of the ERM classifier implying a slight decrease in performance. The original covariance values for different combination of features were inappreciable. Thus, assuming that the features are independent did not significantly deviate the results. Nevertheless, the shape implies that the classification model is reasonably good.



Gamma Ideal - 2.333333 and corresponding minimum error 0.019478

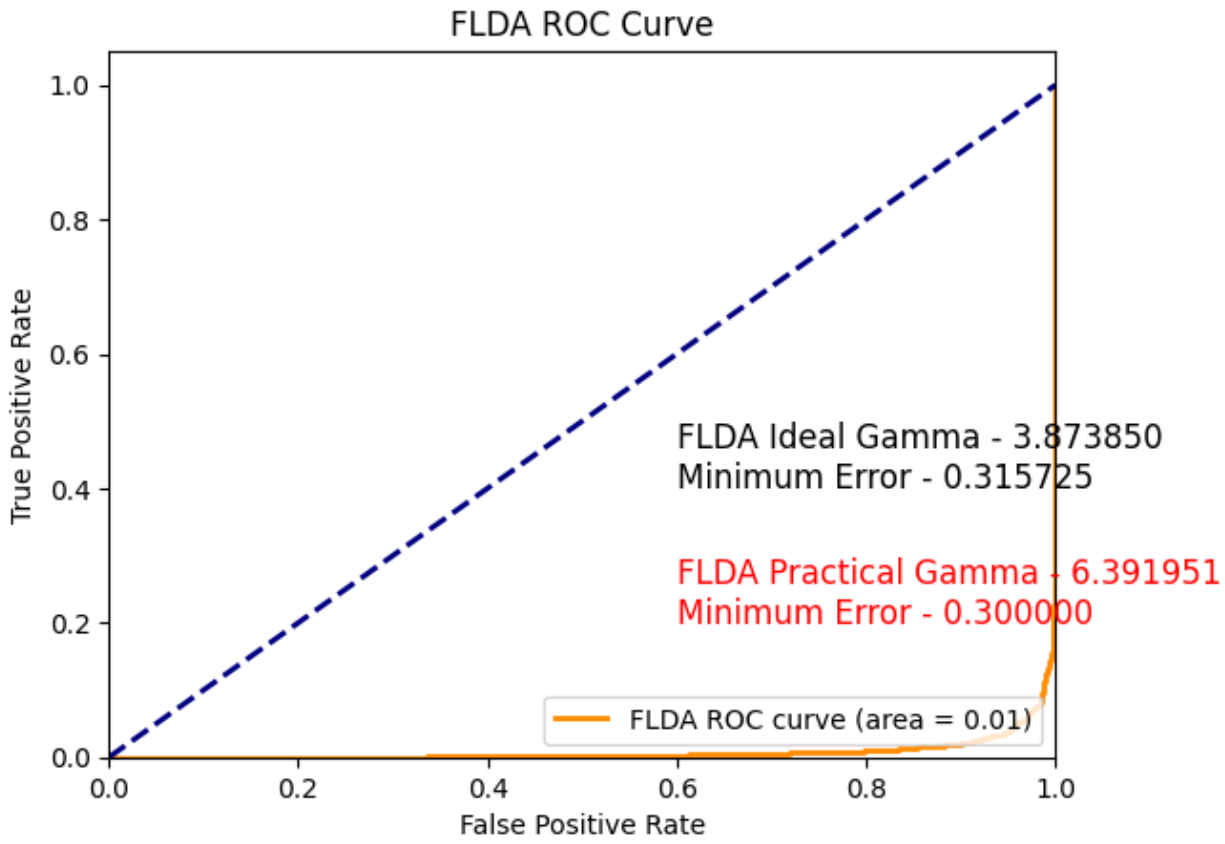
Gamma Practical - 2.175382 and corresponding minimum error 0.019179

Part C – Fisher LDA Classification

1. LDA Classification Rule is as follows:

(D = 1) wTLDA x > τ (D = 0)

2. ROC curve is plotted by ranging τ from minimum to maximum value taking mid-points of consecutive values as thresholds. The shape of the ROC curve implies that the classification model is not up to the mark.



3. It can be observed that there is a notable difference between the theoretical and practical values of γ. The Minimum P error is considerably larger than the previous two Bayesian classifiers possibly due to the overlapping nature of the data distribution

\

Question 2 :

1. Generating Data:

P (x | L = 0) = 0.3 P (x | L = 1) = 0.3 P (x | L = 2) = 0.4

Mean Vectors for Gaussian Mixtures-

Class 0 - μ0 = [0, 0, 15] Gaussian Mixture 1

Class 1 - μ1 = [0, 15, 0] Gaussian Mixture 2

Class 2 – μ2 = [15, 0, 0] with P (Gauss 3 | L = 2) = 0.5

μ3 = [15, 0, 15] with P (Gauss 4 | L = 2) = 0.5

Here, we assume a cube with edge length as 15 units. The 4 corners of the cube are taken as mean vectors for the gaussian distributions to compute the class conditional PDF. The covariance matrices are obtained using the formula given below-

C = (s^2) (I + aA) (I+aA) where |a| << 1

A is a random square matrix with size equal to the number of features, i.e., 3 and s is around 0.4 E. By adding the term aA to the identity matrix, it can be ensured that the distribution is elliptical gaussian with eccentricity a.

2. Decision Rule –

$$\begin{matrix} R(D=0|x) \\ R(D=1|x) \\ R(D=2|x) \end{matrix} = \mathrm{A} * \begin{bmatrix} P(L=0|x) \\ P(L=1|x) \\ P(L=2|x) \end{bmatrix}$$

Risk Loss Matrix Class Posteriors

Decision(x) = argmin Risk (Decision(x) = d | x)

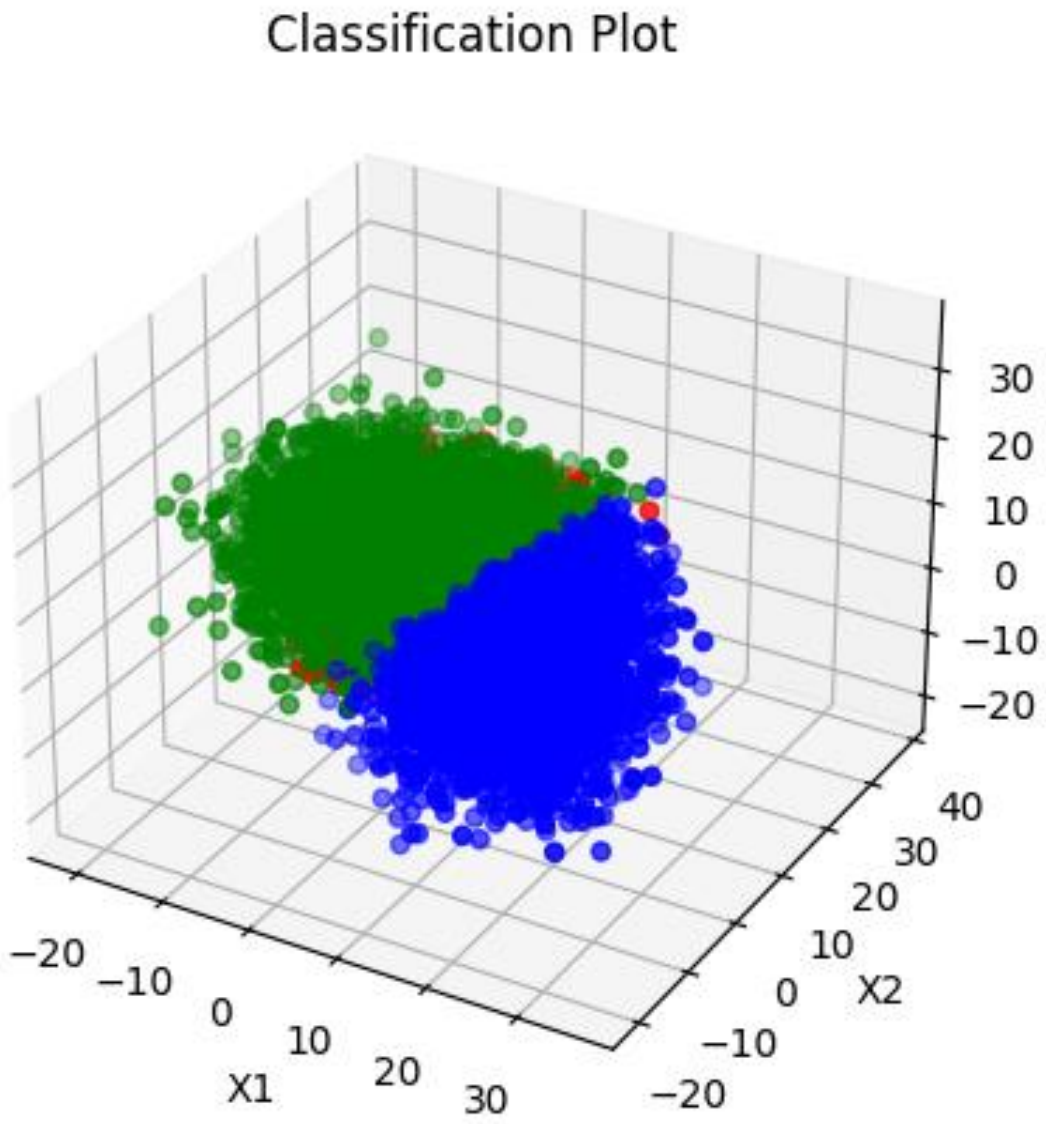
d ε {0,1,2}

3. Visualization 3-D Scatter Plot

```
loss_matrix = np.array([[0, 1, 1], [1, 0, 1], [1, 1, 0]])
```

Average Expected Risk 0.08946848782319178

Confusion Matrix:
[[0.92221511 0.03473862 0.25901304]
[0.03393086 0.92445194 0.02045513]
[0.04385403 0.04080944 0.72053183]]

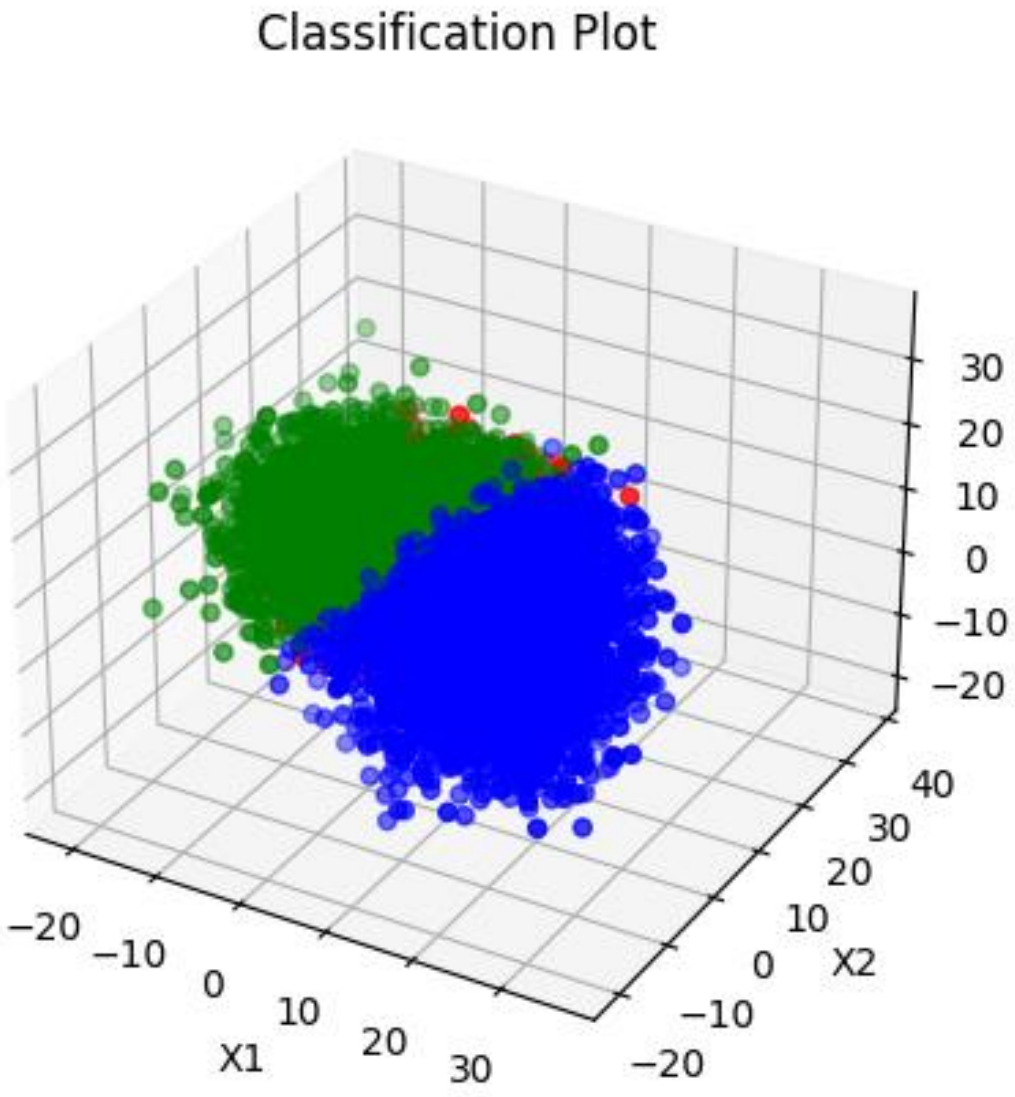


PART B :

loss_matrix = np.array([[0, 10, 10], [1, 0, 10], [1, 1, 0]])

Average Expected Risk 0.23828798409420457

Confusion Matrix:
[[0.76536492 0.00337268 0.12119662]
[0.06338028 0.81281619 0.00485809]
[0.1712548 0.18381113 0.87394528]]

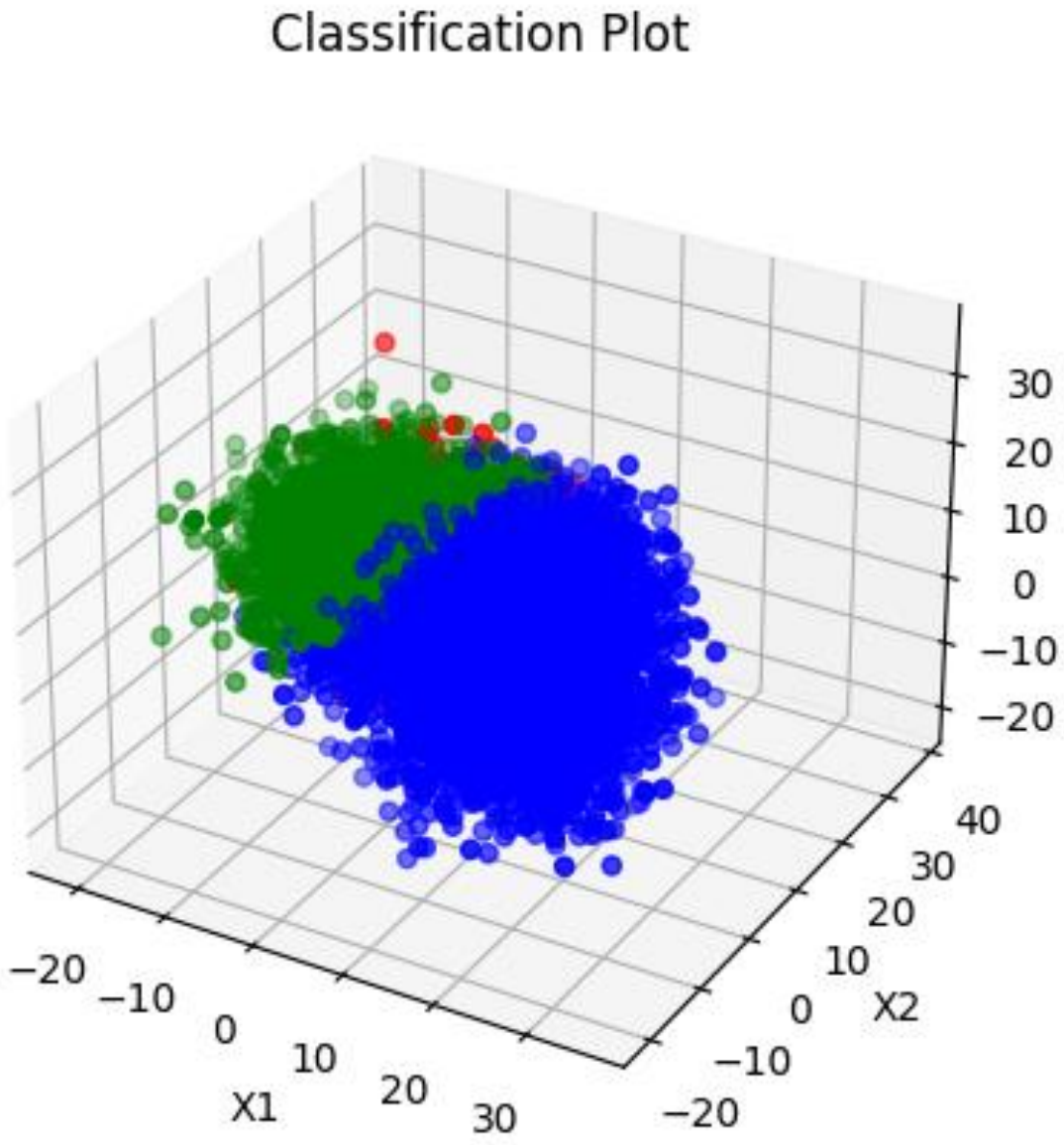


loss_matrix = np.array([[0, 100, 100], [1, 0, 100], [1, 1, 0]])

Average Expected Risk 0.4204203584141175

Confusion Matrix:
[[5.02880922e-01 3.37268128e-04 4.27000767e-02]
[3.84122919e-02 5.68634064e-01 1.02275633e-03]
[4.58706786e-01 4.31028668e-01 9.56277167e-01]]

As penalty for label 3 is increased, the percentage of misclassifications decreases for this class. With higher loss for a label, the model gets effectively trained to avoid making wrong decisions for this label by compromising other decisions.



Question 3 :

PART A :

Dataset Link - https://archive.ics.uci.edu/ml/datasets/Wine+Quality.

The Red Wine Dataset file was used to train the model.

From the distribution it is evident that samples 5 and 6 tend to dominate the dataset whereas samples 0,1,2,9 and 10 are not present in the training set. The dataset consists of 11 features which is a reasonable number to train a computationally efficient model. Any kind of dimensionality reduction technique may result in loss of significant information. Therefore, it is considered all features are relevant to build the classification model.

For the class conditional PDF to be gaussian for the given features, it is assumed that thesamples are independent and identically distributed along with 1600 samples to be a

sufficiently large number to apply the central limit theorem. This theorem states that a sample distribution approximates a normal distribution if the number of samples is large enough.

Using the sample count, the class priors are computed using the following formula:

P(L) = Numbers of samples belonging to class L / Total Number of samples

On testing the conditionality of the co-variance matrices, it is identified that majority of them yield a very large conditional number. Therefore, it is essential to add a small regularization value to broaden the distribution.

$C_{Regularized} = C_{SampleAverage} + \lambda I$

Here, λ is a hyper-parameter which decides the amount of regularization added to the

original variance values. To calculate λ, I considered finding out the arithmetic average of the non zero eigen values of the matrix. The trace (sum of diagonal elements) is equal to the sum of eigen values of a matrix and rank is number of non-zero eigen values.

Hence,

Arithmetic Average = trace ($C_{SampleAverage}$) / rank ($C_{SampleAverage}$)

$\lambda = \alpha$ (Arithmetic Average)

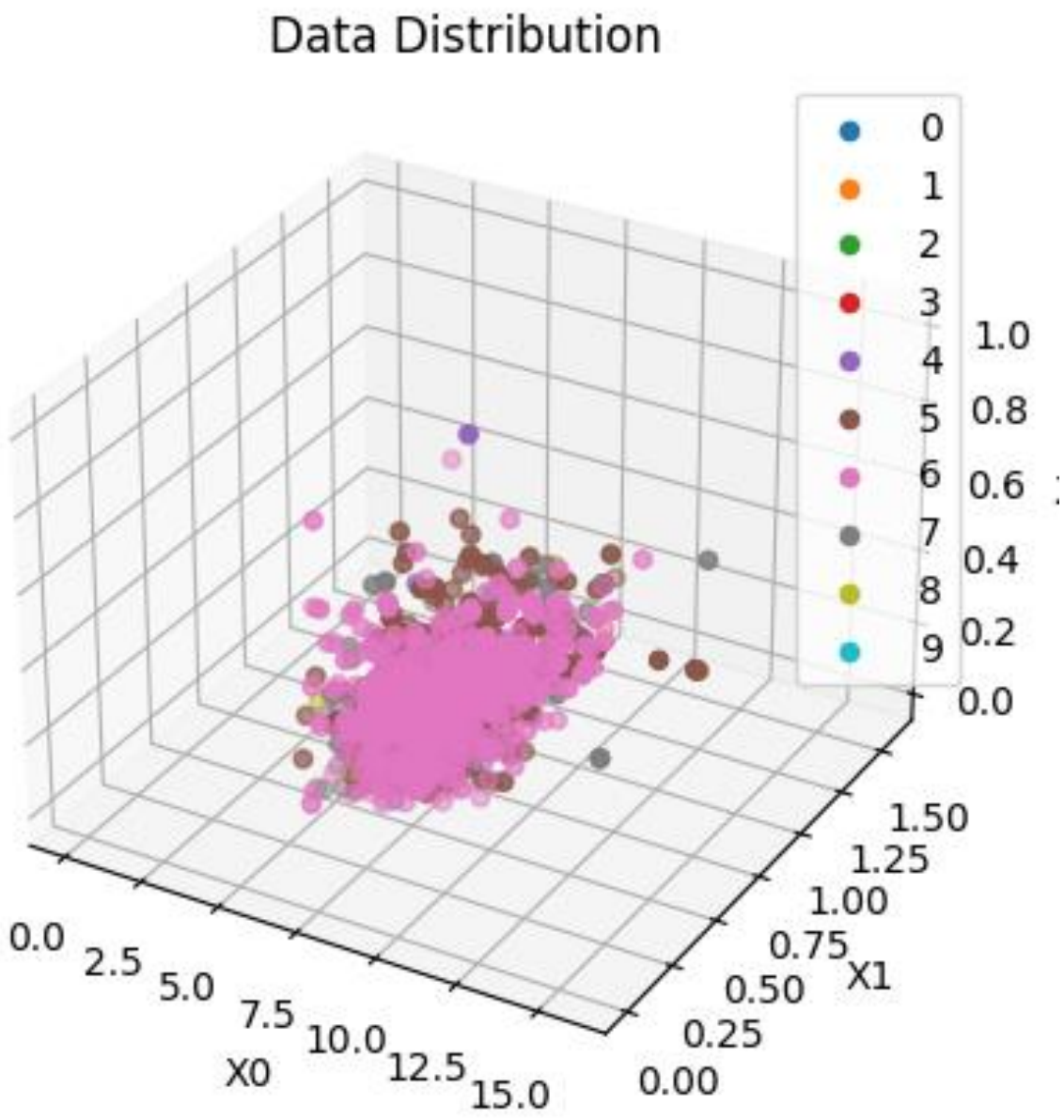
where α is a small real number between 0 and 1. This is a hyperparameter controlling the main hyperparameter λ, which can be tuned in the range of 10-3 to 10-9 to check for maximum accuracy of the model. The loss function selected was 0-1 loss in order to allot equal penalty for all the incorrect decisions and 0 for correct decisions. Thus, a MAP classifier is designed to solve this classification problem.

Minimum Expected Risk 0.32440311787912685

Confusion Matrix C =

```
[[0.         0.         0.         0.         0.         0.         0.         0.         0.         0.
]
 [0.         0.         0.         0.         0.         0.         0.         0.         0.         0.
]
 [0.         0.         0.         0.         0.         0.         0.         0.         0.         0.
]
 [0.         0.         0.         1.         0.         0.         0.         0.         0.         0.
]
 [0.         0.         0.         0.         0.18867925 0.0190     0.0156     0.00502    0.         0.
]
 [0.         0.         0.         0.         0.43396226 0.6328     0.2210     0.0251     0.         0.
]
 [0.         0.         0.         0.         0.33962264 0.3259     0.6473     0.4472     0.1667     0.
]
 [0.         0.         0.         0.         0.03773585 0.0220     0.0971     0.4522     0.         0.
]
 [0.         0.         0.         0.         0.         0.         0.0188     0.0703     0.8333     0.
]
 [0.         0.         0.         0.         0.         0.         0.         0.         0.         0.
]]
```

The results suggest that using Gaussian models for classification might not always be the best choice, especially when we lack domain-specific knowledge about the problem. It's possible that the class posteriors were strongly influenced by the priors, which were estimated based on sample counts. This influence could potentially lead to misleading results when applied to new samples. Therefore, in real-world scenarios, it's crucial to carefully select prior beliefs to strike a balance between priors and the importance of the features, such as the class-conditional PDF.



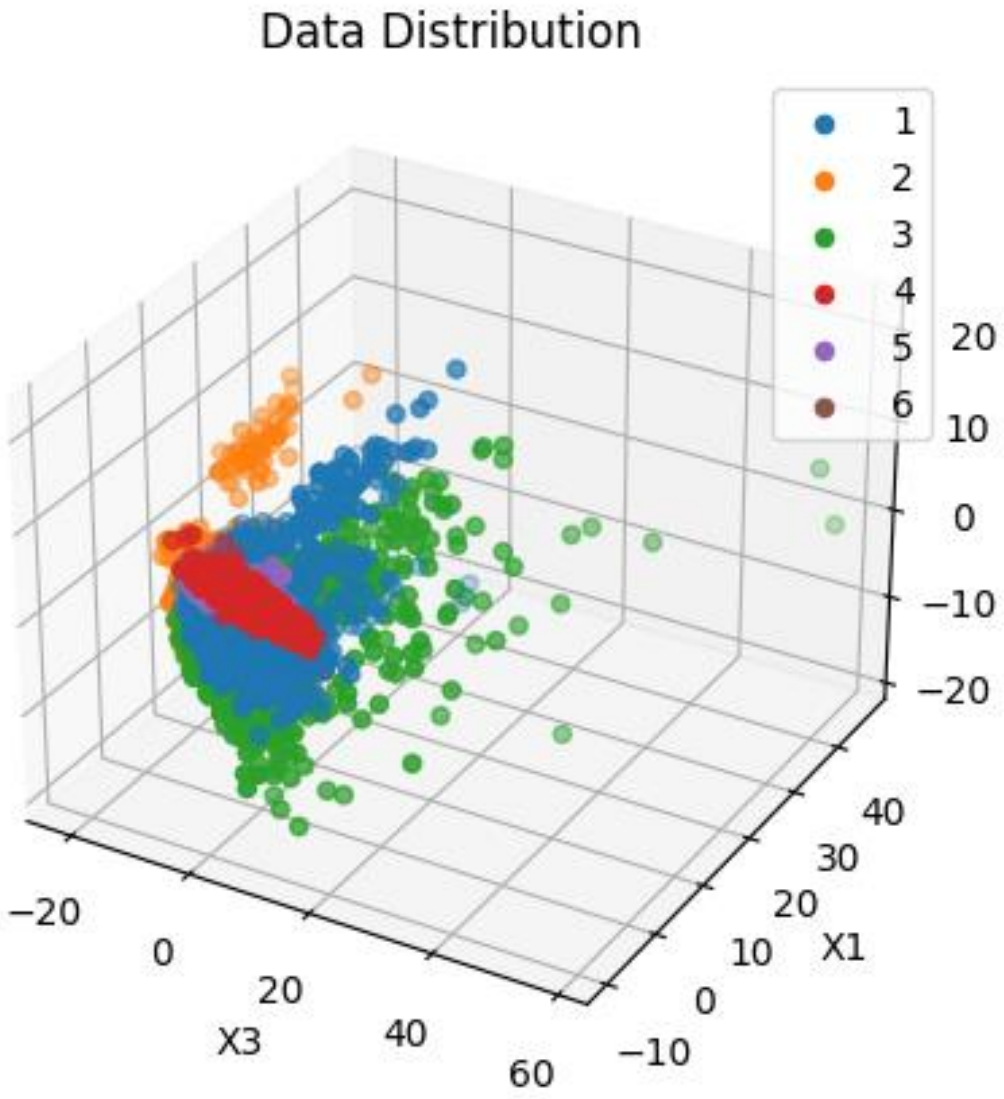
In practical terms, it was observed that choosing a very small value (e.g., in the range of 10^-9) for a hyperparameter, denoted as α, led to the best model performance. This finding highlights that an excessive value for this hyperparameter can hinder the model from capturing essential distribution details, emphasizing the importance of its appropriate selection.

PART B :

Dataset Link https://archive.ics.uci.edu/ml/datasets/Human+Activity+Recognition+Using+Smartphones

The dataset contains 561 features, which can lead to various issues such as redundant features that do not significantly contribute to the classification task, as well as correlated features that provide similar information. Moreover, handling a large number of features can result in increased computation time. To address these challenges, Principal Component Analysis (PCA) is employed as a dimensionality reduction technique. PCA aims to transform the original features into a smaller set of more relevant ones by maximizing the variance explained.

Data visualization after applying PCA :



Unique labels in 'label' array: [1 2 3 4 5 6]
Unique labels in 'y_train.txt' file: [1 2 3 4 5 6]

Minimum Expected Risk 0.091187269194214

```
C =   [[0.912   0.0242 0.0294   0.        0.0014      0.   ]
      [0.0562  0.9198 0.1267 0.0015       0.        0.   ]
      [0.0318  0.0559 0.8438   0.         0.       0.0007]
      [0.         0.     0.   0.6041      0.0800     0.0326]
      [0.         0.     0.   0.3545      0.9155      0.   ]
      [0.         0.     0.   0.0396      0.0029    0.9665]]
```

From the data distribution and class priors calculated, it is evident that the data is evenly distributed between all the classes. The regularization parameter and assumptions regarding class-conditional PDF are similar to the previous part.

In this case, a Gaussian classifier performs better than in the Wine Quality Classification problem. This is because the dataset is well-balanced, meaning that the different classes have similar proportions. As a result, the model can focus on the patterns in the data and relationships between the reduced set of features and the class labels. PCA, a dimensionality reduction technique, helps extract useful information from the large number of features. However, it's essential to test this model with new data to ensure it doesn't perform too well on the training data but poorly on unseen samples, which would indicate overfitting.

APPENDIX

Question 1 :

PART A :

```
from turtle import color
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
from mpl_toolkits.mplot3d import Axes3D

np.set_printoptions(threshold=np.inf)
plt.rcParams['figure.figsize'] = [9, 9]
N_features = 4  # Number of features
N_Samples = 10000  # Number of Samples
N_labels = 2  # Number of classes

# Mean vectors
mean_matrix = np.ones(shape=[N_labels, N_features])
mean_matrix[0, :] = [-1, -1, -1, -1]

# Covariance matrices
covariance_matrix = np.ones(shape=[N_labels, N_features, N_features])
covariance_matrix[0, :, :] = [[2, -0.5, 0.3, 0], [-0.5, 1, -0.5, 0], [0.3, -0.5, 1, 0], [0, 0, 0, 2]]
covariance_matrix[1, :, :] = [[1, 0.3, -0.2, 0], [0.3, 2, 0.3, 0], [-0.2, 0.3, 1, 0], [0, 0, 0, 3]]

# Seed to obtain the same results for random numbers
np.random.seed(10)

# Class Priors and assigning labels
priors = [0.7, 0.3]
label = (np.random.rand(N_Samples) >= priors[1]).astype(int)

# Generate Gaussian distribution for 10,000 samples using mean and covariance matrices for each label
X = np.zeros(shape=[N_Samples, N_features])
for i in range(N_Samples):
    if label[i] == 0:
        X[i, :] = np.random.multivariate_normal(mean_matrix[0, :], covariance_matrix[0, :, :])
    elif label[i] == 1:
        X[i, :] = np.random.multivariate_normal(mean_matrix[1, :], covariance_matrix[1, :, :])
```

```
# Compute discriminant score using class conditional PDF
GaussPDF0 = np.log(multivariate_normal.pdf(X, mean=mean_matrix[0, :], cov=np.eye(N_features)))
GaussPDF1 = np.log(multivariate_normal.pdf(X, mean=mean_matrix[1, :], cov=np.eye(N_features)))
discrim_score = GaussPDF1 - GaussPDF0

# Sort tau values to navigate from the minimum to the maximum value
sorted_tau = np.sort(discrim_score)
tau_sweep = []

# Calculate mid-points which will be used as threshold values
for i in range(0, 9999):
    tau_sweep.append((sorted_tau[i] + sorted_tau[i + 1]) / 2.0)

# Array initialization for results
decision = []
TP = [None] * len(tau_sweep)
FP = [None] * len(tau_sweep)
minPerror = [None] * len(tau_sweep)

# Classify for each threshold and compute error and evaluation metrics
for (index, tau) in enumerate(tau_sweep):
    decision = (discrim_score >= tau)
    TP[index] = (np.size(np.where((decision == 1) & (label == 1))) / np.size(np.where(label == 1)))
    FP[index] = (np.size(np.where((decision == 1) & (label == 0))) / np.size(np.where(label == 0)))
    minPerror[index] = (priors[0] * FP[index]) + (priors[1] * (1 - TP[index]))

# Theoretical classification based on class priors
loggamma_ideal = np.log(priors[0] / priors[1])
ideal_decision = (discrim_score >= loggamma_ideal)
TP_ideal = (np.size(np.where((ideal_decision == 1) & (label == 1))) / np.size(np.where(label == 1)))
FP_ideal = (np.size(np.where((ideal_decision == 1) & (label == 0))) / np.size(np.where(label == 0)))
minPerror_ideal = (priors[0] * FP_ideal) + (priors[1] * (1 - TP_ideal))
print("Gamma Ideal - %f and corresponding minimum error %f" % (np.exp(loggamma_ideal), minPerror_ideal))
print("Gamma Practical - %f and corresponding minimum error %f" %(np.exp(tau_sweep[np.argmin(minPerror)]), np.min(minPerror)))

# Plot ROC curve
plt.plot(FP, TP, color='blue')
```

```
plt.xlabel('False Positive')
plt.ylabel('True Positive')
plt.title('ROC Curve')
plt.plot(FP[np.argmin(minPerror)], TP[np.argmin(minPerror)], 'o', color='black')

# Plot Data Distribution
fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")
Class0 = ax.scatter(X[label == 0, 3], X[label == 0, 1], X[label == 0, 2], '+', color='green',
label="0")
Class1 = ax.scatter(X[label == 1, 3], X[label == 1, 1], X[label == 1, 2], '.', color='blue',
label="1")
plt.xlabel('X3')
plt.ylabel('X1')
ax.set_zlabel('X2')
ax.legend()
plt.title('Data Distribution')
plt.show()
```

PART B :

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal

np.set_printoptions(threshold=np.inf)
plt.rcParams['figure.figsize'] = [9, 9]

N_features = 4  # Number of features
N_Samples = 10000  # Number of Samples
N_labels = 2  # Number of classes

# Mean vectors
mean_matrix = np.ones(shape=[N_labels, N_features])
mean_matrix[0, :] = [-1, -1, -1, -1]

# Covariance matrices
covariance_matrix = np.ones(shape=[N_labels, N_features, N_features])
covariance_matrix[0, :, :] = [[2, -0.5, 0.3, 0],
                              [-0.5, 1, -0.5, 0],
                              [0.3, -0.5, 1, 0],
                              [0, 0, 0, 2]]
```

```
covariance_matrix[1, :, :] = [[1, 0.3, -0.2, 0],
                              [0.3, 2, 0.3, 0],
                              [-0.2, 0.3, 1, 0],
                              [0, 0, 0, 3]]

# Seed to obtain the same results for random numbers
np.random.seed(10)

# Class Priors and assigning labels
priors = [0.7, 0.3]
label = (np.random.rand(N_Samples) >= priors[1]).astype(int)

# Generate Gaussian distribution for 10000 samples using mean and covariance matrices
for each label
X = np.zeros(shape=[N_Samples, N_features])
for i in range(N_Samples):
    if label[i] == 0:
        X[i, :] = np.random.multivariate_normal(mean_matrix[0, :], np.eye(N_features))
    elif label[i] == 1:
        X[i, :] = np.random.multivariate_normal(mean_matrix[1, :], np.eye(N_features))

# Compute discriminant score using class conditional PDF
GaussPDF0 = np.log(multivariate_normal.pdf(X, mean=mean_matrix[0, :],
cov=np.eye(N_features)))
GaussPDF1 = np.log(multivariate_normal.pdf(X, mean=mean_matrix[1, :],
cov=np.eye(N_features)))
discrim_score = GaussPDF1 - GaussPDF0

# Sort tau values to navigate from minimum to maximum value
sorted_tau = np.sort(discrim_score)
tau_sweep = []

# Calculate mid-points which will be used as threshold values
for i in range(0, 9999):
    tau_sweep.append((sorted_tau[i] + sorted_tau[i + 1]) / 2.0)

# Array initialization for results
decision = []
TP = [None] * len(tau_sweep)
FP = [None] * len(tau_sweep)
minPerror = [None] * len(tau_sweep)

# Classify for each threshold and compute error and evaluation metrics
for (index, tau) in enumerate(tau_sweep):
    decision = (discrim_score >= tau)
```

```
    TP[index] = (np.size(np.where((decision == 1) & (label == 1))) / np.size(np.where(label == 1)))
    FP[index] = (np.size(np.where((decision == 1) & (label == 0))) / np.size(np.where(label == 0)))
    minPerror[index] = (priors[0] * FP[index]) + (priors[1] * (1 - TP[index]))

# Theoretical classification based on class priors
loggamma_ideal = np.log(priors[0] / priors[1])
ideal_decision = (discrim_score >= loggamma_ideal)
TP_ideal = (np.size(np.where((ideal_decision == 1) & (label == 1))) / np.size(np.where(label == 1)))
FP_ideal = (np.size(np.where((ideal_decision == 1) & (label == 0))) / np.size(np.where(label == 0)))
minPerror_ideal = (priors[0] * FP_ideal) + (priors[1] * (1 - TP_ideal))

print("Gamma Ideal - %f and corresponding minimum error %f" % (np.exp(loggamma_ideal), minPerror_ideal))

# Plot ROC curve
plt.plot(FP, TP, color='blue')
plt.xlabel('False Positive')
plt.ylabel('True Positive')
plt.title('ROC Curve')
plt.plot(FP[np.argmin(minPerror)], TP[np.argmin(minPerror)], 'o', color='black')
plt.show()

print("Gamma Practical - %f and corresponding minimum error %f" % (np.exp(tau_sweep[np.argmin(minPerror)]), np.min(minPerror)))
```

PART C :

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
from sklearn.metrics import roc_curve, auc
from sklearn.model_selection import train_test_split

np.random.seed(10)

N_features = 4
N_Samples = 10000
N_labels = 2

mean_matrix = np.ones(shape=[N_labels, N_features])
```

```
mean_matrix[0, :] = [-1, -1, -1, -1]

covariance_matrix = np.ones(shape=[N_labels, N_features, N_features])
covariance_matrix[0, :, :] = [[2, -0.5, 0.3, 0], [-0.5, 1, -0.5, 0], [0.3, -0.5, 1, 0], [0, 0, 0, 2]]
covariance_matrix[1, :, :] = [[1, 0.3, -0.2, 0], [0.3, 2, 0.3, 0], [-0.2, 0.3, 1, 0], [0, 0, 0, 3]]

priors = [0.7, 0.3]
label = (np.random.rand(N_Samples) >= priors[1]).astype(int)

X = np.zeros(shape=[N_Samples, N_features])
for i in range(N_Samples):
    if label[i] == 0:
        X[i, :] = np.random.multivariate_normal(mean_matrix[0, :], covariance_matrix[0, :, :])
    elif label[i] == 1:
        X[i, :] = np.random.multivariate_normal(mean_matrix[1, :], covariance_matrix[1, :, :])

X_train, X_test, y_train, y_test = train_test_split(X, label, test_size=0.5)

# Fisher Linear Discriminant Analysis (FLDA)
mean_class0 = np.mean(X_train[y_train == 0], axis=0)
mean_class1 = np.mean(X_train[y_train == 1], axis=0)

Sb = np.outer(mean_class0 - mean_class1, mean_class0 - mean_class1)
Sw = np.cov(X_train[y_train == 0], rowvar=False) + np.cov(X_train[y_train == 1], rowvar=False)

# Calculate FLDA projection vector
eigenvalues, eigenvectors = np.linalg.eig(np.linalg.inv(Sw).dot(Sb))
projection_vector = eigenvectors[:, np.argmax(eigenvalues)]

# FLDA projections
X_train_flda = X_train.dot(projection_vector)
X_test_flda = X_test.dot(projection_vector)

# Threshold for FLDA
threshold = np.mean(X_train_flda[y_train == 0]) - np.mean(X_train_flda[y_train == 1])

# FLDA ROC curve
fpr_flda, tpr_flda, thresholds_flda = roc_curve(y_test, X_test_flda)
roc_auc_flda = auc(fpr_flda, tpr_flda)

# Plot ROC curve
plt.figure()
plt.plot(fpr_flda, tpr_flda, color='darkorange', lw=2, label='FLDA ROC curve (area = %0.2f)' % roc_auc_flda)
```

```
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('FLDA ROC Curve')
plt.legend(loc='lower right')

# Calculate practical and ideal tau values and corresponding minimum errors
gamma_ideal = threshold
ideal_decision = (X_test_flda >= gamma_ideal)
TP_ideal = np.sum((ideal_decision == 1) & (y_test == 1)) / np.sum(y_test == 1)
FP_ideal = np.sum((ideal_decision == 1) & (y_test == 0)) / np.sum(y_test == 0)
minPerror_ideal = (priors[0] * FP_ideal) + (priors[1] * (1 - TP_ideal))

min_error_idx_flda = np.argmin((priors[0] * fpr_flda + priors[1] * (1 - tpr_flda)))
tau_practical_flda = thresholds_flda[min_error_idx_flda]
minPerror_practical_flda = priors[0] * fpr_flda[min_error_idx_flda] + priors[1] * (1 - tpr_flda[min_error_idx_flda])

plt.text(0.6, 0.4, f"FLDA Ideal Gamma - {gamma_ideal:.6f}\nMinimum Error - {minPerror_ideal:.6f}", fontsize=12)
plt.text(0.6, 0.2, f"FLDA Practical Gamma - {tau_practical_flda:.6f}\nMinimum Error - {minPerror_practical_flda:.6f}", fontsize=12, color='red')

plt.show()
```

Question 2 :

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal

np.random.seed(10)

N = 10000
N_features = 3
N_labels = 3
N_mixtures = 4
```

```
priors = np.array([0.3, 0.3, 0.4])

mean_matrix = np.array([[0, 0, 15], [0, 15, 0], [15, 0, 0], [15, 0, 15]])

covariance_matrix = np.zeros(shape=[N_mixtures, N_features, N_features])
for i in range(N_mixtures):
    scale_factor = 0.01 * (i + 1)
        covariance_matrix[i, :, :] = 36 * np.linalg.matrix_power(np.eye(N_features) +
(scale_factor * np.random.randn(N_features, N_features)), 2)

randomlabels = np.random.rand(N)
label = np.zeros(N)

cumsum = np.cumsum(priors)
for i in range(N):
    for j in range(N_labels):
        if randomlabels[i] <= cumsum[j]:
            label[i] = j
            break

X = np.zeros(shape=[N, N_features])
for i in range(N):
    if label[i] == 0:
        X[i, :] = np.random.multivariate_normal(mean_matrix[0, :], covariance_matrix[0, :, :])
    elif label[i] == 1:
        X[i, :] = np.random.multivariate_normal(mean_matrix[1, :], covariance_matrix[1, :, :])
    else:
        mixture = 2 if np.random.rand(1, 1) >= 0.5 else 3
                    X[i, :] = np.random.multivariate_normal(mean_matrix[mixture, :],
covariance_matrix[mixture, :, :])

loss_matrix = np.array([[0, 1, 1], [1, 0, 1], [1, 1, 0]])

P_x_given_L = np.zeros(shape=[N_labels, N])
for i in range(N_labels):
          P_x_given_L[i, :] = multivariate_normal.pdf(X, mean=mean_matrix[i, :],
cov=covariance_matrix[i, :, :])

P_x = np.matmul(priors, P_x_given_L)
# Calculate Class Posteriors
ClassPosteriors = (P_x_given_L * np.matlib.repmat(priors, N, 1).T) /
np.matlib.repmat(P_x, N_labels, 1)

ExpectedRisk = np.matmul(loss_matrix, ClassPosteriors)
Decision = np.argmin(ExpectedRisk, axis=0)
```

```
ConfusionMatrix = np.zeros(shape=[N_labels, N_labels])
for d in range(N_labels):
    for l in range(N_labels):
            ConfusionMatrix[d, l] = (np.size(np.where((d == Decision) & (l == label))) / np.size(np.where(label == l)))

fig = plt.figure()
ax = plt.axes(projection="3d")
colors = ['g', 'r', 'b']

for d in range(N_labels):
    for l in range(N_labels):
        indices = (label == d) & (Decision == l)
        ax.scatter(X[indices, 0], X[indices, 1], X[indices, 2], color=colors[l], marker='o')

plt.xlabel('X1')
plt.ylabel('X2')
ax.set_zlabel('X3')
plt.title('Classification Plot')
plt.show()

print("Average Expected Risk", np.sum(np.min(ExpectedRisk, axis=0) / N))
print("Confusion Matrix:\n", ConfusionMatrix)
```

Question 3 :

Dataset 1 :

```
import numpy as np
from scipy.stats import multivariate_normal
import pandas as pd
from numpy import linalg as LA
import matplotlib.pyplot as plt

# Load the dataset
url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-red.csv'
df = pd.read_csv(url, delimiter=';')
Data = df.values

N = Data.shape[0]      # Number of Samples
N_labels = 10          # Number of classes (adjust this based on your dataset)
```

```
# Extract class labels from the last column (assumes it's the last column)
label = Data[:, -1]
Data = Data[:, :-1]  # Feature set

N_features = Data.shape[1]  # Number of features

mean_matrix = np.zeros(shape=[N_labels, N_features])
covariance_matrix = np.zeros(shape=[N_labels, N_features, N_features])

# Compute Mean Vectors and Covariance matrices
for i in range(0, N_labels):
    if i in label:
        mean_matrix[i, :] = np.mean(Data[label == i], axis=0)
        covariance_matrix[i, :, :] = np.cov(Data[label == i], rowvar=False)
         covariance_matrix[i, :, :] += (0.000000005) * ((np.trace(covariance_matrix[i, :, :])) /
LA.matrix_rank(covariance_matrix[i, :, :])) * np.eye(N_features)

# Assign 0-1 loss matrix
loss_matrix = np.ones(shape=[N_labels, N_labels]) - np.eye(N_labels)

# Compute class conditional PDF
P_x_given_L = np.zeros(shape=[N_labels, N])
for i in range(0, N_labels):
    if i in label:
            P_x_given_L[i,  :]  =  multivariate_normal.pdf(Data,  mean=mean_matrix[i,  :],
cov=covariance_matrix[i, :, :])

# Estimate class priors based on sample count
priors = np.zeros(shape=[N_labels, 1])
for i in range(0, N_labels):
    if i in label:
        priors[i] = np.sum(label == i) / N

# Compute Class Posteriors using priors and class conditional PDF
P_x = np.dot(priors.T, P_x_given_L)
ClassPosteriors = (P_x_given_L * (np.tile(priors, (1, N)) / np.tile(P_x, (N_labels, 1))))

# Evaluate Expected risk and decisions based on minimum risk
ExpectedRisk = np.dot(loss_matrix, ClassPosteriors)
Decision = np.argmin(ExpectedRisk, axis=0)
print("Average Expected Risk", np.sum(np.min(ExpectedRisk, axis=0)) / N)

# Estimate Confusion Matrix
ConfusionMatrix = np.zeros(shape=[N_labels, N_labels])
for d in range(N_labels):
```

```
    for l in range(N_labels):
        if l in label and d in label:
            ConfusionMatrix[d, l] = np.sum((d == Decision) & (l == label)) / np.sum(label == l)

print(ConfusionMatrix)

# Plot Data Distribution (you may need to choose a subset of features for 2D or 3D visualization)
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
for i in range(N_labels):
    ax.scatter(Data[(label == i), 0], Data[(label == i), 1], Data[(label == i), 2], label=i)
plt.xlabel('X0')
plt.ylabel('X1')
ax.set_zlabel('X2')
ax.legend()
plt.title('Data Distribution')
plt.show()
```

PART B :

```
import random
import numpy as np
import numpy.matlib
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
from mpl_toolkits.mplot3d import Axes3D
import pandas as pd
from numpy import linalg as LA
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

from google.colab import files

uploaded_file_path = '/content/train.csv'
# Import Dataset
df = pd.read_csv(uploaded_file_path)
Data = df.to_numpy()

# Identifying labels and size of the dataset
N = Data.shape[0]

uploaded_file_path_1 = '/content/y_train.txt'
Y = pd.read_csv(uploaded_file_path_1)
```

```
label = np.squeeze(Y.to_numpy())

# Debugging: Check unique labels in 'label' array and 'y_train.txt' file
unique_labels = np.unique(label)
print("Unique labels in 'label' array:", unique_labels)

unique_labels_y_train = np.unique(np.squeeze(Y.to_numpy()))
print("Unique labels in 'y_train.txt' file:", unique_labels_y_train)

# Normalizing data to apply PCA
Data = Data[:, 0:-2]
sc = StandardScaler()
Data = sc.fit_transform(Data)

# Reducing dimensions to obtain 10 principal components
pca = PCA(n_components=10)
Data = pca.fit_transform(Data)

N_labels = 6  # Number of labels
N_features = 10  # Number of features

mean_matrix = np.zeros(shape=[N_labels, N_features])
covariance_matrix = np.zeros(shape=[N_labels, N_features, N_features])

# Compute Mean Vectors and Covariance matrices
for i in range(N_labels):
    mean_matrix[i, :] = np.mean(Data[label == (i + 1), :], axis=0)
    covariance_matrix[i, :, :] = np.cov(Data[label == (i + 1), :], rowvar=False)
      covariance_matrix[i, :, :] += (0.00001) * ((np.trace(covariance_matrix[i, :, :])) /
np.linalg.matrix_rank(covariance_matrix[i, :, :])) * np.eye(10)

# Assign 0-1 loss matrix
loss_matrix = np.ones(shape=[N_labels, N_labels]) - np.eye(N_labels)

# Compute class conditional PDF
P_x_given_L = np.zeros(shape=[N_labels, Data.shape[0]])

for i in range(N_labels):
        P_x_given_L[i, :] = multivariate_normal.pdf(Data, mean=mean_matrix[i, :],
cov=covariance_matrix[i, :, :])

# Estimate class priors based on sample count
priors = np.zeros(shape=[N_labels, 1])
for i in range(N_labels):
    priors[i] = np.sum(label == (i + 1)) / Data.shape[0]
```

```
# Compute Class Posteriors using priors and class conditional PDF
P_x = np.dot(priors.T, P_x_given_L)
ClassPosteriors = (P_x_given_L * (np.tile(priors, (1, Data.shape[0])) / np.tile(P_x, (N_labels, 1))))

# Evaluate Expected risk and decisions based on minimum risk
ExpectedRisk = np.dot(loss_matrix, ClassPosteriors)
Decision = np.argmin(ExpectedRisk, axis=0)
print("Average Expected Risk", np.sum(np.min(ExpectedRisk, axis=0)) / Data.shape[0])

# Estimate Confusion Matrix
ConfusionMatrix = np.zeros(shape=[N_labels, N_labels])

for d in range(N_labels):
    for l in range(N_labels):
        ConfusionMatrix[d, l] = np.sum((d == Decision) & (l == (label - 1))) / np.sum(label - 1 == l)

np.set_printoptions(suppress=True)
print(ConfusionMatrix)

# Plot Data Distribution
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
for i in range(1, N_labels + 1):
    ax.scatter(Data[label == i, 1], Data[label == i, 2], Data[label == i, 3], label=i)

plt.xlabel('X3')
plt.ylabel('X1')
ax.set_zlabel('X2')
ax.legend()
plt.title('Data Distribution')
plt.show()
```